Final Project - RIO 203

François BURLACOT
francois.burlacot@telecom-paristech.fr

Axel TORCHY
axel.torchy@telecom-paristech.fr

Christophe VUONG
christophe.vuong@telecom-paristech.fr

February 22, 2019

Contents

Part I

Designing an IoT application

1 Introduction

Our IoT application aims at modelizing some aspects of a drinks dispenser. Ironically, we did not focus on any aspect with regards to the dispensing of the drinks, but we chose some other aspects that might be related to a (imaginary) drinks dispenser in order to implement only some functionalities so as to illustrate a few ideas regarding the Internet of Things: communication between sensors using CoAP protocol, Contiki, FIT/IoT-lab, etc.

The functionalities are rather simple. The dispenser uses a few sensors:

- a **button** (for example, it could be replace by a movement sensor to determine whether a customer is standing in front of the machine, or could be triggered by the first press on the numeric pad used to choose the drink) to turn the dispenser on;
- another **button** used for maintenance to display the temperature value inside the dispenser;
- a **light sensor** in order to switch the lights on if the luminosity in the room becomes too low, or to switch the lights off if it is high enough;
- an **accelerometer sensor** used to check if someone is trying to shake the dispenser in order to make a drink fall. It displays a warning message on the screen, but it could also call the maintenance staff or security staff in order to deal with the situation;
- a **temperature sensor**, the value from which is displayed periodically (every 30 seconds) on the screen of the dispenser in order to inform the customer that the drinks are kept cool. Its information is also used by the previous *second button sensor* when the maintainance staff wants to have the information withouth having to wait for 30 seconds.

2 Description of the architecture

We used the FIT/IoT-LAB platform, based on the `04-er-rest-example` client and server codes that we adapted to our application. Hence, we also used the `RPL Border Router` original code (in `examples/ipv6/rpl-border-router/border-router.c`).

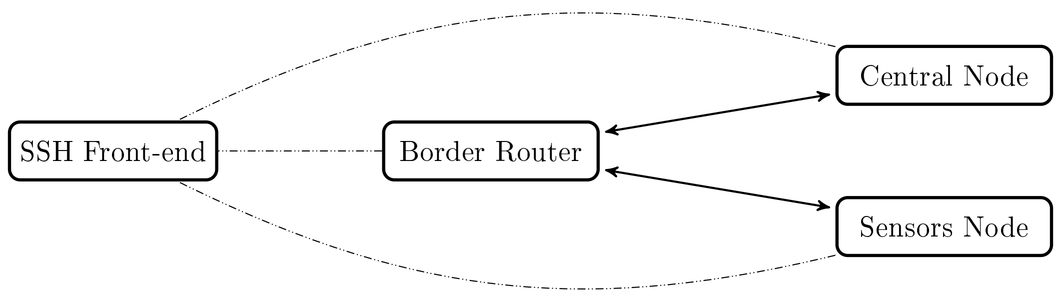


Figure 1: Architecture of our FIT/IoT-LAB experiment.

Through **SSH Frontend** we could flash the firmware onto the nodes, and used the `tunslip` and `nc` commands.

Depending on the routes, there might be a link between **Central Node** and **Sensors Node**.

The **Central Node** represents the actual dispenser that is interacting with the customer, whereas the **Sensors Node** represents a hidden part of the system that is only communicating with the **Central Node** in order to exchange informations and trigger reactions from this latter.

3 Detail of the implementation

For our implementation, we created some new CoAP resources, accessible either via a `GET` or `POST` CoAP request. The code is mainly divided in two parts: one part for the **Central Node** and one part for the **Sensors Node**.

3.1 Implementation of the Central Node

This node is the one which interacts with the customer. It possesses a few additional CoAP resources, including **proactive alarms**, that can be triggered by CoAP `POST` requests from the **Sensors Node**.

The code is accessible in the file `central_node.c`.

Here are the new resources that we created for the **Central Node**:

- `res_button_alarm`: when accessed via a `POST` CoAP message, this resource displays a welcoming message on the screen to encourage the customer to choose a drink.

  The code can be found in the file:
  `resources/res-button-alarm.c`

- `res_lighton_alarm`: when accessed via a `POST` CoAP message, this resource displays a message saying that the luminosity is too **low** and **switches on the light**.

  The code can be found in the file:
  `resources/res-lighton-alarm.c`

- `res_lightoff_alarm`: when accessed via a `POST` CoAP message, this resource displays a message saying that the luminosity is too **high** and **switches off the light**.

  The code can be found in the file:
  `resources/res-lightoff-alarm.c`

- `res_shaking_alarm`: when accessed via a `POST` CoAP message, this resource displays **a message warning the user not to shake the dispenser**. It could also for example call the security staff in order to stop this violent behavior.

  The code can be found in the file:
  `resources/res-shaking-alarm.c`

- `res_temperature_update`: when accessed via a `POST` CoAP message which *payload* contains a value of temperature, this resource displays the value of temperature in the dispenser sent by the **Sensors Node** on the screen. This update is periodically sent (every 30 seconds in our application).

  The code can be found in the file:
  `resources/res-temperature-update.c`

The code of the **Central Node** requires both a **CoAP Server** (in order to receive the `POST` messages) and a **CoAP Client** (in order to send a `GET` request whenever the button to display temperature is pressed on the dispenser. Thus, there are two Contiki processes (using the `PROCESS_THREAD` macro).

The client part of the **Central Node** sends its requests to the **Sensors Node**. It is thus mandatory to change the IPv6 address in the source code *before compiling* to the **Sensors Node**'s IPv6 address.

3.2 Implementation of the Sensors Node

The **Sensors Node** utilizes the **light sensor**, the **temperature sensor**, the **accelerometer sensor** and a **button sensor** which triggers the welcoming message.

We only created one supplementary resource:

- `res_my_temperature`: when accessed via a `GET` CoAP message, this resource sends back the value of the temperature sensor to the **Central Node**.

  The code can be found in the file:
  `resources/res-my-temperature.c`

The client part of the **Sensors Node** sends its requests to the **Central Node** node. It is thus mandatory to change the IPv6 address in the source code *before compiling* to the **Central Node**'s IPv6 address.

4 Before the demo

The drawback of using the FIT/IoT-LAB is that we have no control on the sensors' values. It becomes complicated to test the application because there is no way to trigger events that could activate a reaction of one of the nodes (*e.g.* sending a `POST`/`GET` request to the other node).

For example, it is impossible to change the light in the room where the sensors are located to make sure that the **Sensors Node** sends a `POST` CoAP request to the **Central Node**'s **res_lighton_res** resource if the light gets under the threshold that has been fixed beforehand.

To solve this problem, we conditionnally changed our code (using `C` preprocessor directives `#ifdef TEST_PURPOSE` and `#define TEST_PURPOSE` in both **Central Node** and **Sensors Node** code. The idea is simple: we wanted to **use the console serial port** of the sensors in order to simulate sensors events such as a change in the light/accelerometer value, the button pressed, etc. The remaining of the behavior is the same as it would be with our "initial" implementation.

After flashing the firmware on the nodes, you must then open two new terminals on the SSH Front-end and use the `nc m3-XXX 20000` command.

Here is how to use this **test purpose** implementation :

- On the **Central Node** :
  - `b` triggers a **button pressed event**.
- On the **Sensors Node** :
  - `l XXXX` triggers a **light sensor event** with the light value `XXXX` which must be an integer value;
  - `b` triggers a **button pressed event**;
  - `s` triggers a **shaking event** (which normally would be triggered by the accelerometer value being above the defined threshold).

5 Demo

In our demo, we used three nodes on the Lille site:

Node function	Node id	Firmware	IPv6 address
Border Router	m3-32	border-router.c	2001:660:4403:483::3358
Central Node	m3-34	central_node.c	2001:660:4403:483::8773
Sensors Node	m3-36	sensors_node.c	2001:660:4403:483::9577

We modified the Makefile: we changed the dependencies of the `all` target and replaced them with `central_node` and `sensors_node` (the source file names must match these names) in order to compile easily using the `make TARGET=iotlab-m3` command.

5.1 Periodic update of the temperature inside the dispenser

The **Sensors Node** sends a CoAP `POST` request to the **Central Node** every 30 seconds. This value is displayed on the screen of the dispenser on the **Central Node**.

Note: to implement this feature, we used an `etimer` as can be seen in the source code `sensors_node.c`

Here is the output on the **Central Node** and **Sensors Node** consoles:

Central Node:

```
[PERIODIC UPDATE] The temperature inside the dispenser is:  4190864
[PERIODIC UPDATE] The temperature inside the dispenser is:  4191201
[PERIODIC UPDATE] The temperature inside the dispenser is:  4191434
```

Sensors Node:

```
Temperature sent to server.  Value=4190864
Temperature sent to server.  Value=4191201
Temperature sent to server.  Value=4191434
```

We notice that the period between each message is around 30 seconds, as was expected.

5.2 Lights alarm when the luminosity in the room changes

We use the `l XXXX` command on the **Sensors Node** console in order to trigger light events with value `XXXX`, and see what happens on the console on the **Central Node**.

Note: the threshold of luminosity to switch the light on/off was arbitrarily fixed to `100`

Sensors Node:

```
l 80
-- Room is too dark: light_sensor value=80. Alarm sent to switch on light.
|-- Done. --
l 80
|-- Done. --
l 100
|-- Done. --
Temperature sent to server. Value=4193175
|
--Done--
l 101
-- Room is bright enough: light_sensor value=101. Alarm sent to switch off light
|-- Done. --
```

Central Node:

```
The room is too dark. The light will be switched on.
[PERIODIC UPDATE] The temperature inside the dispenser is: 4193175
The room is bright enough. The light swill be switched off.
```

The feature works according to what was expected. We notice that in this implementation, the alarm is sent only if the state changes, for example if the light before was **above** the threshold, and the new value is **below** the threshold.

5.3 Welcoming message when the customer presses the button

We trigger a **button pressed event** on the **Sensors Node** using the test `b` command on the console. Here is what is then displayed on the **Central Node**'s console:

```
Hello customer! you pressed the button. please choose the drink you wish.
 [ALARM_BUTTON]
```

From this and the previous section, we are ensured that the CoAP communication between the two nodes works fine.

5.4 Temperature displayed on the screen if requested by maintainance staff

Let's try to request the temperature value without having to wait for the periodic update in order to display it on the **Central Node**'s screen. We do it using the test `b` command on the console:

```
b
-- Button pressed. Requesting temperature to sensor. Please wait. --
The temperature inside the dispensor is: 4190887
```

The response to the request comes with an acceptable delay (less than half a second). There is no output on the **Sensors Node**'s console, because there is no `printf` command in the `resource/res-my-temperature.c` source code of the resource.

5.5 Shaking alarm if the dispenser is being shaken

Let's try to simulate an accelerometer event with a value above the accelerometer threshold, using the `s` (for shaking) command on the **Sensors Node**'s console:

```
s
-- The dispenser has been shaken. An alarm will be sent.
|-- Done. --
```

Almost immediately, the output on the **Central Node**'s console is:

```
PLEASE DO NOT SHAKE THE DISPENSER. NO FREE DRINKS WILL COME OUT OF IT!
```

Hence, this feature also works as was expected. We could imagine to implement another reaction from the **Central Node** in addition to this simple warning message, such as an e-mail sent to the security staff, etc.

6 Conclusion

All the console ouput lines above were copied and pasted from the terminal using the `nc` command from the SSH front-end (two different terminal for each of the nodes).

The same results can be obtained by following theses steps:

- Include the `central_node.c` and `sensors_node.c` source files in the `04-er-rest-example` folder;
- Change the first line of the `Makefile` to this:
  `all: central_node sensors_node`
- **Don't forget to change the IPv6 addresses in both source codes!** (the **Central Node**'s address in the **Sensors Node**'s code, and vice versa).
- Compile everything using `make TARGET=iotlab-m3`
- Flash the Border Router image on a first node;
- Flash the `central_node.iotlab-m3` and `sensors_node.iotlab-m3` images on two other nodes not far away on the same website (we used Lille);
- Launch in two different terminals from the SSH frontend the `nc m3-XXX` command for the two last nodes in order to monitor the console outputs;
- Test everything using the test-purpose commands as described previously.
- Enjoy!

Our only regret is that we weren't able to test it in real condition with the real sensors' events. We tried using Cooja to be able to control everything about the sensors, the button etc., but we faced problems regarding the `.text` section being too small (the code was probably too large), that is why we used FIT/IoT-LAB instead.

Part II

Question 1: Energy efficiency problem

Here is our network topology (in green, the server; in yellow, the clients):

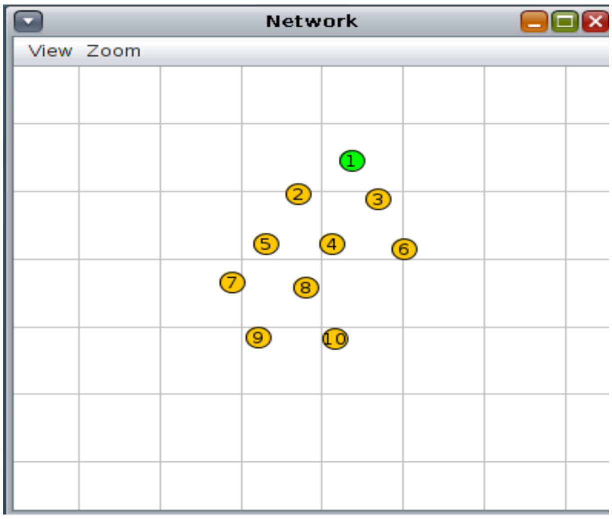


Figure 2: Network topology.

The connections between the nodes are as follows:

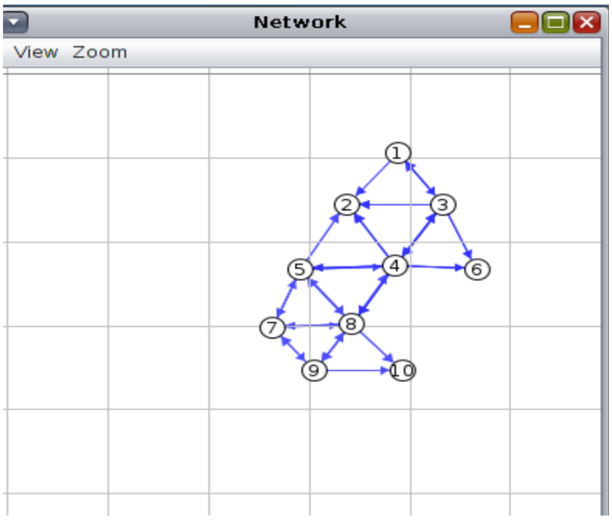


Figure 3: Network topology.

1 Test of UDP with agregation

We used the following code:

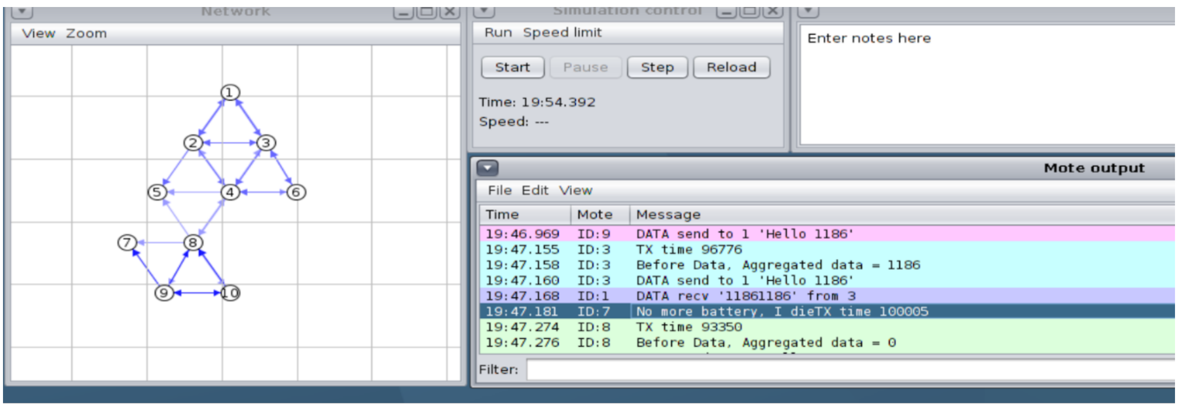
```
static void
send_packet(void *ptr)
{ static int seq_id;
  char buf [MAX_PAYLOAD_LEN];
  unsigned long tx_time;         //it has the type unsigned long to be able to
  // reach 100 000
  tx_time = energest_type_time (ENERGEST_TYPE_TRANSMIT);
```

```
  printf("TX time %4lu\n",tx_time);

  if (tx_time < (unsigned long) 100000){
  //below energy limit, the mote still has some battery and is transmitting
    printf("Before Data, Aggregated data = %lu\n", global_reader);
    seq_id++;
    PRINTF("DATA send to %d 'Hello %d'\n",
     server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], seq_id);

    if (global_reader != 0){                //agregation of packets
      sprintf(buf, "%d%lu", seq_id, global_reader);

    }
    else{     //send only his packet
      sprintf(buf, "%d", seq_id);
    }
    uip_udp_packet_sendto(client_conn, buf, strlen(buf),
    &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
     //transmission of packets
  }
  if (tx_time>= (unsigned long) 100000)
  printf("No more battery, I die");
  //if the mote have no energy, He die and stop the transmission
}
```

2 What is the first node to run out of energy? Record its time.

The first mote to die is node number 7 at time `19:47.181` minutes.

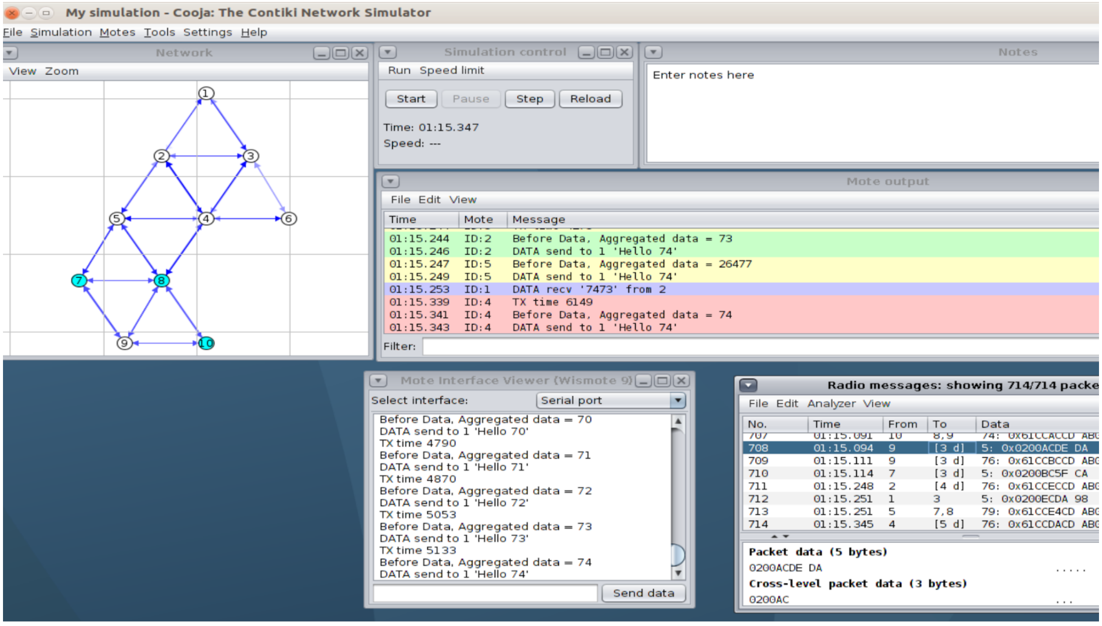
3 When the first node runs out of energy, what happens in the network? Explain in detail, especially related with the used routing protocol and its functions.



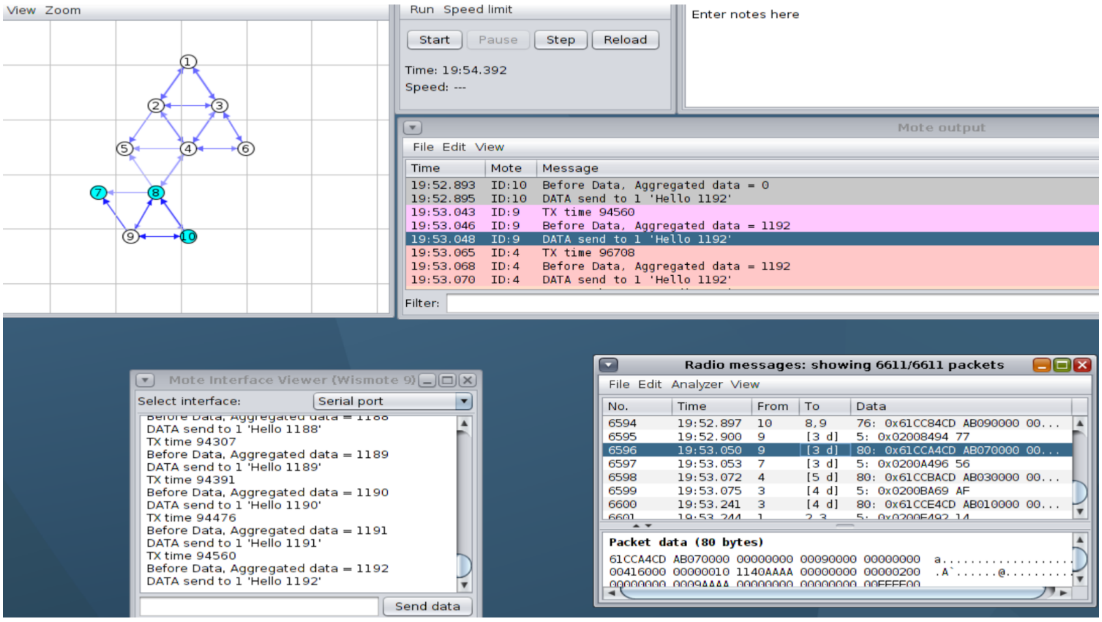
After the mote 7 died, each node who had in is routing table the node 7 don't have to find another node to replace it because we are using UDP. Thus, there is not any real acknowledge of all the packets received. If the packets of the node 9 aren't received anymore, he will not know and so will continue to send his packets anyway.

We can see that with the node 9.

Before 7 died, destinations of packets of node 9 (in blue, to whom he is sending packets, called [3 d]):



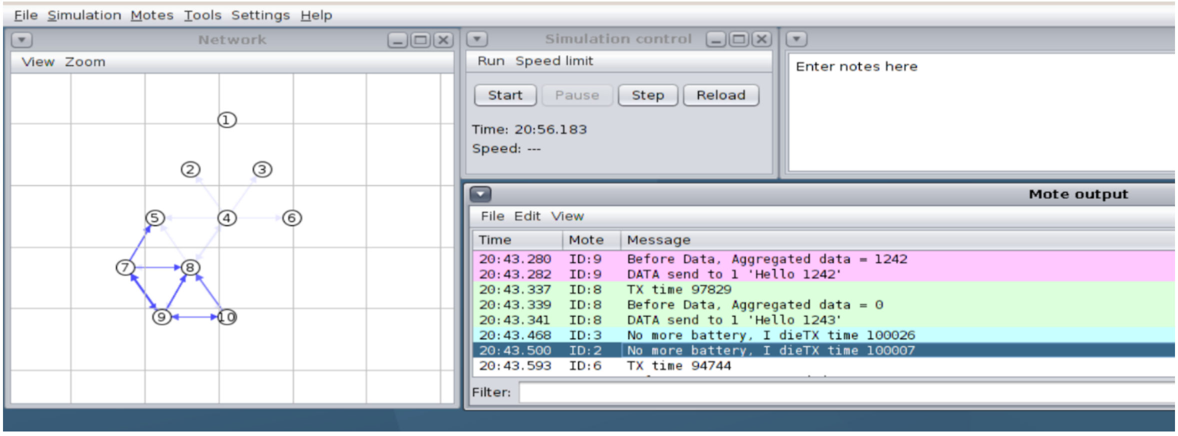
And after node 7 died, this did not change:



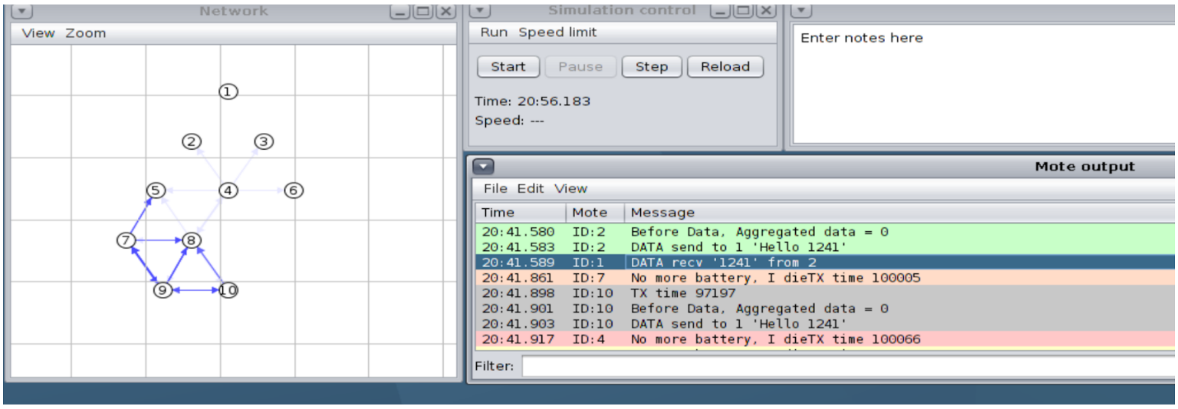
So, like we said, the node 9 is still sending his packets to and via the sames nodes, and because his main route passes through node 7, his packets will not be received anymore (because 7 died).

4 When does the network completely lose its service capabilities?

The server doesn't receive any message when node 3 and 2 died because they are the only nodes who sent messages to server. Thus, when the two are disconnected, the server no longer receives any message and doesn't acknowledge any message of other nodes neither. In our experiment this happened at the time `20:43.500`, when node 2 died (node 3 already died at time `20:29.57`). So the two nodes stopped sending packets to the server.



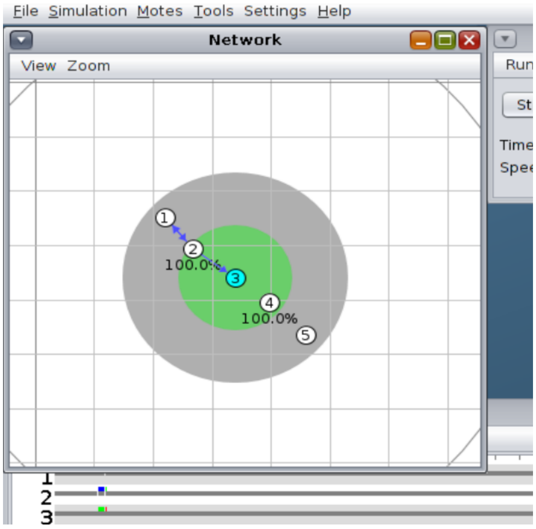
We could also say that the network lose his service when the server(node 1) send his last acknowledge. This happened at time `20:41.589`.



Part III
Question 2

1 Main idea

We use the knowledge we have on the network to build an aggregation model. The most important point is when it is the node's turn to send data.



I interpreted the instructions for the aggregated model as following. Once node 5 has something to send that can be aggregated, node 4 retrieves that and sends aggregated data to node 3 who does the same. Node 2 gathers all the aggregated data, and also sends its data to the server here node 1. Some piece of data could be lost because node 5 emits every three seconds. However, it does not matter since the study is not focused on delivery ratio.

In the multi-hop line topology, we assume that the intermediate nodes need to transmit data when the next node has put data in its uip_buf. In terms of variable, it means that the pointer to `uip_buf` has its content updated.

2 Implementation of aggregation

We need to change the uip6.c file as data in `UIP_BUF` is handled differently compared to the original process without aggregation.

Pieces of data in the buffer are considered as string shared between the `uip6.c` and client file.

Here is the chunk of code in uip6.c that retrieves data without forwarding packet.

```
extern int global_reader;

/* ... */

UIP_IP_BUF->ttl = UIP_IP_BUF->ttl - 1;
      PRINTF("Forwarding packet to ");
      PRINT6ADDR(&UIP_IP_BUF->destipaddr);
      PRINTF("\n");
      UIP_STAT(++uip_stat.ip.forwarded);
      /**handle send or agreggate**/
      remove_ext_hdr();
      char *data_pack = &uip_buf[UIP_IPUDPH_LEN + UIP_LLH_LEN];
      global_reader = atoi(data_pack); //extern variable
```

```
      goto drop;
      //goto send; //non-aggregation
```

Then the udp-client.c file provides code to make the client aggregate data from next to its in the function `send_packet`.

```
static void send_packet(void *ptr)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN];
  static int reader;

  /**Let assume packet forward one by one, easier to measure delay**/
  if (node_id == N_CLIENTS + 1) {
      seq_id++;
      PRINTF("Send 'Hello %d'\n", seq_id);
      sprintf(buf, "%d", seq_id);
      uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                          &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));

  } else {
      if(global_reader!=reader){ //check if data updated in the buffer
        seq_id++;
        PRINTF("DATA send to %d by %d'Hello %d'\n",
          server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], node_id, seq_id);  //to
    be sure of the source of the packet
        PRINTF("Aggregated data = %lu\n",global_reader);
        sprintf(buf, "%d0%d%lu", node_id, seq_id, global_reader);
        uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                          &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
        reader = global_reader;

      }
  }

  /**energy**/
  unsigned long tx_time;
  tx_time = energest_type_time(ENERGEST_TYPE_TRANSMIT);
  PRINTF("Tx Time %lu\n", tx_time);
}
```

3 Limitations of the code

There are other implementations we thought about but they share the same drawbacks as this one and give unexpected results.

Here are the main setbacks of this implementation:

- The integer in the string in the buffer represents the right information as long as it does not go beyond 32 767, otherwise it can be negative. It means that the message in the buffer should not overpass 5 digits. Given the number of nodes, it is impossible with that method to add the node it within the pack of data. The minimum number of digits of the integer is 4 (for data) + 4 (for the id). Of course, separators may be needed to study more parameters.

- The analysis of delay may not be easy at first sight because we did not implement time calculation that would concern the client process and the server process. We chose to run analysis with the text log of mote output in Excel.

4 Analysis

4.1 Energy efficiency

We used the log provided by Cooja and analyse it in a workbook software (Excel). So we can use filters based on ideas and the printed (by `printf`) message to get the TX time value.

At the left without aggregation, at the right with aggregation for the same duration and for the same number of packets sent (around 100).

We do not notice big differences for early packet transmission but after 5 minutes, the difference is there. Even though the aggregation model takes more time to to send packets, it consumes less energy, because of less headers overall for data.

The difference for node 5 is because the size of the message in aggregation model is not exactly the same. Nevertheless as we see below, it is not that significative compared to the difference for the other nodes.

94	04:41.182	ID:5	Tx Time 9102	9102
95	04:45.064	ID:5	Tx Time 9193	9193
96	04:47.291	ID:5	Tx Time 9284	9284
97	04:50.986	ID:5	Tx Time 9375	9375
98	04:53.775	ID:5	Tx Time 9466	9466
99	04:55.783	ID:5	Tx Time 9557	9557
100	05:00.635	ID:5	Tx Time 9648	9648
101	05:03.454	ID:5	Tx Time 9702	9702
102	05:03.868	ID:5	Tx Time 9755	9755
103	05:07.001	ID:5	Tx Time 9809	9809

95	04:44.768	ID:5	Tx Time 8102	8102
96	04:46.994	ID:5	Tx Time 8180	8180
97	04:50.690	ID:5	Tx Time 8258	8258
98	04:53.479	ID:5	Tx Time 8336	8336
99	04:55.486	ID:5	Tx Time 8414	8414
100	05:00.338	ID:5	Tx Time 8492	8492
101	05:03.159	ID:5	Tx Time 8572	8572
102	05:03.573	ID:5	Tx Time 8651	8651
103	05:06.705	ID:5	Tx Time 8731	8731

Figure 4: For node 5

97	04:50.409	ID:4	Tx Time 1653	16533
98	04:52.417	ID:4	Tx Time 1669	16697
99	04:57.268	ID:4	Tx Time 1693	16932
100	05:00.089	ID:4	Tx Time 1702	17023
101	05:00.502	ID:4	Tx Time 1707	17076
102	05:03.634	ID:4	Tx Time 1727	17275

94	04:41.553	ID:4	Tx Time 6448	6448
95	04:43.771	ID:4	Tx Time 6448	6448
96	04:47.475	ID:4	Tx Time 6528	6528
97	04:50.255	ID:4	Tx Time 6528	6528
98	04:52.272	ID:4	Tx Time 6608	6608
99	04:57.124	ID:4	Tx Time 6688	6688
100	04:59.935	ID:4	Tx Time 6688	6688
101	05:00.358	ID:4	Tx Time 6769	6769
102	05:03.491	ID:4	Tx Time 6851	6851
103	05:07.968	ID:4	Tx Time 6933	6933

Figure 5: For node 4

96	04:48.171	ID:3	Tx Time 23629	23629
97	04:50.960	ID:3	Tx Time 23793	23793
98	04:52.968	ID:3	Tx Time 24028	24028
99	04:57.819	ID:3	Tx Time 24335	24335
100	05:00.635	ID:3	Tx Time 24480	24480
101	05:01.053	ID:3	Tx Time 24605	24605
102	05:04.185	ID:3	Tx Time 24877	24877

91	04:33.244	ID:3	Tx Time 6341	6341
92	04:34.804	ID:3	Tx Time 6341	6341
93	04:37.978	ID:3	Tx Time 6424	6424
94	04:41.861	ID:3	Tx Time 6507	6507
95	04:44.078	ID:3	Tx Time 6507	6507
96	04:47.783	ID:3	Tx Time 6590	6590
97	04:50.562	ID:3	Tx Time 6590	6590
98	04:52.580	ID:3	Tx Time 6673	6673
99	04:57.431	ID:3	Tx Time 6756	6756
100	05:00.242	ID:3	Tx Time 6756	6756
101	05:00.666	ID:3	Tx Time 6838	6838
102	05:03.799	ID:3	Tx Time 6922	6922
103	05:08.275	ID:3	Tx Time 7006	7006

Figure 6: For node 3

96	04:47.508	ID:2	Tx Time 30724	**30724**
97	04:50.297	ID:2	Tx Time 30959	**30959**
98	04:52.304	ID:2	Tx Time 31266	**31266**
99	04:57.156	ID:2	Tx Time 31646	**31646**
100	04:59.976	ID:2	Tx Time 31881	**31881**
101	05:00.389	ID:2	Tx Time 32006	**32006**
102	05:03.522	ID:2	Tx Time 32352	**32352**
103				

(a) A subfigure

94	04:41.846	ID:2	Tx Time 6804	**6804**
95	04:44.073	ID:2	Tx Time 6888	**6888**
96	04:47.758	ID:2	Tx Time 6888	**6888**
97	04:50.558	ID:2	Tx Time 6973	**6973**
98	04:52.555	ID:2	Tx Time 6973	**6973**
99	04:57.417	ID:2	Tx Time 7058	**7058**
100	05:00.237	ID:2	Tx Time 7141	**7141**
101	05:00.641	ID:2	Tx Time 7141	**7141**
102	05:03.784	ID:2	Tx Time 7224	**7224**
103	05:08.261	ID:2	Tx Time 7308	**7308**

(b) A subfigure

Figure 7: For node 2

The closer the node is, the more it consumes in classic model compared to the aggregation model, more than four times for node 2. The intermediate nodes in the aggregated model save more than twice as much energy.

4.2 End-to-end delay

In this part we consider the time taken by a packet to go from client to the server. Again, the log from Cooja gives the necessary information to make time calculations. On one hand, without aggregation, it only consists in calculating the difference between the time of the response of the server which announces the node id and the emission by that node. On the other hand, the aggregation model needs that we compute delays from node to node and then cumulate them to obtain the results. We use the approximation of average end-to-end delay equal to the sum of average delays node to node from node i to server. All this was done in Excel, and we got those statistics:

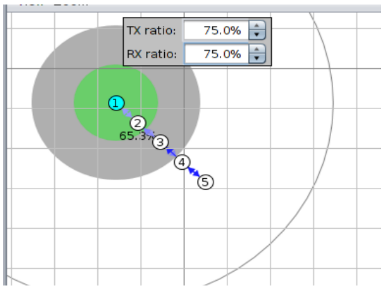
	Node 2	**Node 3**	**Node 4**	**Node 5**
Aggregated	7 ms	2960 ms	3257 ms	4123 ms
Non-aggregated	5.34 ms	9.7 ms	14.1 ms	18.4 ms

These values show how important the increased delay is. It may be problematic in case udp client has to regularly transmit data with a specific period, let say 3 seconds as in our application. However one can notice that end-to-end delays are higher than that period, so some packets may be emitted before previous pack of packet has been received by the server. In a case where the application is interactive-based, for example the client sends sensor data and the server decides what do given the received values and notifies the client. So notifications from the server may not be up to date as it is possible that clients measures new values of light for example whereas the server response is based on previous values.

Part IV
Question 3

1 End-to-end delay

We proceed as previously, but given the risk of lot of packets lost, all our results are only meant to show order of magnitude and ratio between the different values. Indeed the approximation used above for average end-to-end delays is not that accurate here. The average delay node to node do not take into account more or less same number of elements for each node.



It seems that with 5 seconds, packets can take more time to arrive to destination with our model of aggregation.

	Node 2	**Node 3**	**Node 4**	**Node 5**
Aggregated	7 ms	4070 ms	5051 ms	9178 ms
Non-aggregated	5.34 ms	9.7 ms	14.1 ms	18.4 ms

Figure 8: **TX/RX Ratio = 100/100**

	Node 2	**Node 3**	**Node 4**	**Node 5**
Aggregated	7 ms	2708 ms	6391 ms	8664 ms
Non-aggregated	5.34 ms	10 ms	15 ms	21 ms

Figure 9: **TX/RX Ratio = 75/75**

	Node 2	**Node 3**	**Node 4**	**Node 5**
Aggregated	7 ms	2800 ms	6000 ms	9000 ms
Non-aggregated	5.34 ms	10 ms	15 ms	21 ms

Figure 10: **TX/RX Ratio = 50/50**

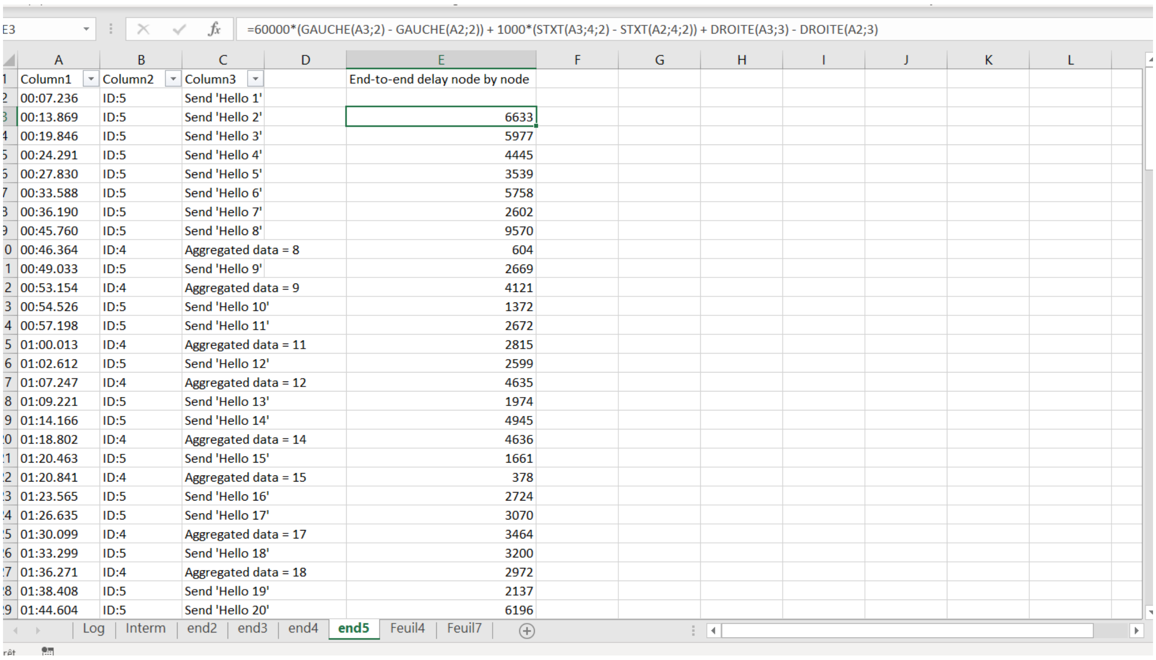
In terms of end-to-end delay, change of TX/RX radio does not really change the ratio of end-to-end delays of the aggregated model and the non-aggregated model. However, since the number of packets decreases for both models when the TX / RX ratio is low, the average end-to-end delay is not that meaningful and this analysis is not precise for this reason.

2 Packet delivery ratio

For aggregated model, we just count the number of packets from each node in the log and the number of packet received by the server during 5 minutes. It gives an estimation because we cannot know exactly at the end of recording if one node is transmitting multi-hop data.

$Ratio_i aggregation = \frac{\text{number of packet received by the server}}{\text{number of packets from node } i}$

Here is a screenshot of how we did to compute it.


E3 =60000*(GAUCHE(A3;2) - GAUCHE(A2;2)) + 1000*(STXT(A3;4;2) - STXT(A2;4;2)) + DROITE(A3;3) - DROITE(A2;3)

Column1	Column2	Column3	End-to-end delay node by node
00:07.236	ID:5	Send 'Hello 1'	
00:13.869	ID:5	Send 'Hello 2'	6633
00:19.846	ID:5	Send 'Hello 3'	5977
00:24.291	ID:5	Send 'Hello 4'	4445
00:27.830	ID:5	Send 'Hello 5'	3539
00:33.588	ID:5	Send 'Hello 6'	5758
00:36.190	ID:5	Send 'Hello 7'	2602
00:45.760	ID:5	Send 'Hello 8'	9570
00:46.364	ID:4	Aggregated data = 8	604
00:49.033	ID:5	Send 'Hello 9'	2669
00:53.154	ID:4	Aggregated data = 9	4121
00:54.526	ID:5	Send 'Hello 10'	1372
00:57.198	ID:5	Send 'Hello 11'	2672
01:00.013	ID:4	Aggregated data = 11	2815
01:02.612	ID:5	Send 'Hello 12'	2599
01:07.247	ID:4	Aggregated data = 12	4635
01:09.221	ID:5	Send 'Hello 13'	1974
01:14.166	ID:5	Send 'Hello 14'	4945
01:18.802	ID:4	Aggregated data = 14	4636
01:20.463	ID:5	Send 'Hello 15'	1661
01:20.841	ID:4	Aggregated data = 15	378
01:23.565	ID:5	Send 'Hello 16'	2724
01:26.635	ID:5	Send 'Hello 17'	3070
01:30.099	ID:4	Aggregated data = 17	3464
01:33.299	ID:5	Send 'Hello 18'	3200
01:36.271	ID:4	Aggregated data = 18	2972
01:38.408	ID:5	Send 'Hello 19'	2137
01:44.604	ID:5	Send 'Hello 20'	6196

Without aggregation, we also use the message as a tag to count the number of packets received by node 1 from node i (figure on the next page) :

	A	B	C
1	Column1	Column2	Column3
106	00:39.547	ID:1	DATA recv '12 from client' from 5
139	00:49.187	ID:1	DATA recv '16 from client' from 5
173	00:58.586	ID:1	DATA recv '19 from client' from 5
197	01:06.703	ID:1	DATA recv '21 from client' from 5
210	01:09.625	ID:1	DATA recv '22 from client' from 5
215	01:09.914	ID:1	DATA recv '23 from client' from 5
242	01:21.094	ID:1	DATA recv '26 from client' from 5
263	01:27.617	ID:1	DATA recv '28 from client' from 5
285	01:32.602	ID:1	DATA recv '30 from client' from 5
301	01:37.961	ID:1	DATA recv '32 from client' from 5
322	01:43.578	ID:1	DATA recv '34 from client' from 5
356	01:52.289	ID:1	DATA recv '37 from client' from 5
398	02:06.383	ID:1	DATA recv '41 from client' from 5
446	02:20.336	ID:1	DATA recv '46 from client' from 5
521	02:43.844	ID:1	DATA recv '54 from client' from 5
551	02:54.367	ID:1	DATA recv '57 from client' from 5
640	03:21.250	ID:1	DATA recv '66 from client' from 5
658	03:27.047	ID:1	DATA recv '68 from client' from 5
739	03:52.086	ID:1	DATA recv '77 from client' from 5
843	04:24.508	ID:1	DATA recv '87 from client' from 5
952	04:58.797	ID:1	DATA recv '99 from client' from 5
967	05:03.648	ID:1	DATA recv '100 from client' from 5
1005	05:14.491	ID:1	DATA recv '104 from client' from 5
1027	05:21.476	ID:1	DATA recv '106 from client' from 5
1032	05:23.734	ID:1	DATA recv '107 from client' from 5
1119	05:49.023	ID:1	DATA recv '116 from client' from 5
1143	05:56.257	ID:1	DATA recv '118 from client' from 5
1176	06:06.312	ID:1	DATA recv '121 from client' from 5

Feuil1 | node5 | 5 | node 4

29 enregistrement(s) trouvé(s) sur 1229

Here are the packet delivery ratios for different TX/RX rates:

	Node 2	Node 3	Node 4	Node 5
Aggregated	100 % (46/46)	94 % (46/49)	88 % (46/52)	75 % (46/61)
Non-aggregated	100 %	100 %	100 %	100 %

Figure 11: **TX/RX Ratio = 100/100**

	Node 2	Node 3	Node 4	Node 5
Aggregated	64 % (11/17)	37 % (11/29)	21 % (11/52)	11 % (11/97)
Non-aggregated	67 % (86/127)	46 % (59/127)	32 % (41/127)	22 % (29/127)

Figure 12: **TX/RX Ratio = 75/75**

	Node 2	Node 3	Node 4	Node 5
Aggregated	33 % (2/6)	12.5 % (2/16)	6 % (2/34)	2.22 % (2/87)
Non-aggregated	49 % (67/135)	25 % (34/134)	9 % (12/134)	4 % (5/134)

Figure 13: **TX/RX Ratio = 50/50**

Here we can see that aggregated model causes more loss of packet. It is striking for the nodes far from server which barely manages to transmit something to the server. Without aggregation it is still OK for 2-hop but when the delivery packet ratio is lower than 50%, it doesn't really matter how low it is. It is not reliable. So aggregation model may be worse in those conditions but it is not worth trying to work in those conditions. Therefore, it is not a big drawback as it may sound.

3 Conclusion

Advantages of aggregation:

- Less headers
- Save energy in large proportion, especially in multi-hop environment

Drawbacks:

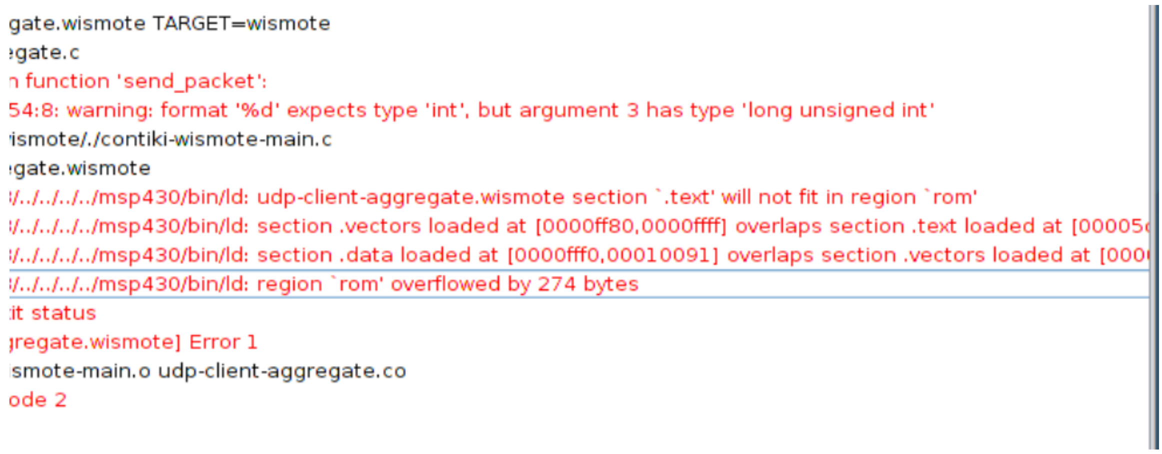
- Increase end-to-end delays until overpassing node period sending
- Suffer a lot more when conditions of network are bad whereas without aggregation it remains OK
- Not sure to receive all the packets

4 Appendix

4.1 Issues concerning `node_id` to concatenate in aggregated data

We did a simplified version as we had several ideas that did not work as good as we expected. It was really constraining not to work with `node_id` because we needed to do approximation to compute average end-to-end delays given the fact we could not easily track a packet coming from node 5 when the number of packets sent is high.

Given the limitation with extern variable as int (`global_reader`), we thought of parsing data into unsigned long. We can use unsigned long variable (`global_reader` in our code) to store the aggregated data since that type of variable has a limit of digits. However as we tried that solution, we met another unexpected issue which came from the compiler.



```
gate.wismote TARGET=wismote
gate.c
n function 'send_packet':
54:8: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
ismote/./contiki-wismote-main.c
gate.wismote
/../../../../msp430/bin/ld: udp-client-aggregate.wismote section `.text' will not fit in region `rom'
/../../../../msp430/bin/ld: section .vectors loaded at [0000ff80,0000ffff] overlaps section .text loaded at [00005
/../../../../msp430/bin/ld: section .data loaded at [0000fff0,00010091] overlaps section .vectors loaded at [000
/../../../../msp430/bin/ld: region `rom' overflowed by 274 bytes
it status
gregate.wismote] Error 1
smote-main.o udp-client-aggregate.co
ode 2
```

It seems that we cannot do much to solve it. We made the code as compact as possible we removed printf to see, but the issue remained.

It does not give the expected results when we use a `char *` character pointer to the content of `uip_buf` in `udp-client` file as an extern variable shared with `uip6.c` which can be sprintf into the buffer as it seems as the most natural way to do aggregation as some tokens appeared (except separators used) in data out of nowhere.